Factors Impacting the Inputs of Traceability Recovery Approaches

Nasir Ali, Yann-Gaël Guéhéneuc, and Giuliano Antoniol

Abstract In requirement engineering, researchers have proposed various tractability recovery approaches. To the best of our knowledge, all traceability recovery approaches have low precision and recall. Our main claim in this chapter is that there exist factors that impact the traceability approaches' inputs, in particular *source document*, *target document*, and *experts' opinion*, that cause low precision and recall. In this chapter, we pursue four objectives: first, to identify and document factors that impact traceability recovery approaches' inputs; second, to identify metrics/tools to measure/improve the quality of the inputs with respect to the identified factors, third, to provide precautions to control these factors, and, fourth, to empirically prove and quantify the effect of one of these factors—expert's programming knowledge—on the traceability recovery approaches' inputs. To achieve the first two objectives, we perform an incremental literature review of traceability recovery approaches and identify and document three key inputs and the seven factors impacting these inputs, out of 12 identified factors. We analyse the reported results in literature for the identified factors to address our third objective. We conduct an empirical study to assess the impact of expert's programming knowledge, to address our fourth objective. We use the effort, number of correct answers, and time to measure the effect of expert's programming knowledge on traceability recovery. We conclude that, in the literature, seven factors impacting the inputs of traceability recovery approaches have been identified, documented, and reported along with related metrics/tools and precautions. We suggest that practitioners should be wary of these seven factors and researchers should focus on the five others to improve traceability recovery approaches.

Nasir Ali, Yann-Gaël Guéhéneuc, and Giuliano Antoniol
DGIGL, École Polytechnique de Montréal, Canada,
e-mail: {nasir.ali,yann-gael.gueheneuc}@polymtl.ca,~antoniol@ieee.org

Table 1 Average precision and recall range of TRAs, bold values represent the example presented in 1

	VSM		**LSI**		**JS**		**Rule-based**	
Data Sets	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
SCA [1]	20 – 43	51 – 76	14 – 26	41 – 78	23 – 41	57 – 78	–	–
CORBA [1]	50 – **80**	68 – 89	11 – 50	14 – 61	43 – 65	55 – 81	–	–
MODIS [68]	**7.9**	75.6	4.2 – 6.3	63.4 – 92.6	–	–	–	–
CM-1 [68]	**1.5**	97.7	**0.9**	98.6 – **98.8**	–	–	–	–
Easy Clinic [61]	17 – 80	4 – 90	17 – 60	3 – 90	17 – 80	4 – 91	–	–
eTour [61]	17 – 68	5 – 47	17 – 64	4 – 46	17 – 76	5 – 47	–	–
Mobile Phone [41]	–	–	–	–	–	–	81 – 95.9	65.4 – 97.2
UCMS, TV Software [67]	–	–	–	–	–	–	60 – 81	68 – 85

1 Introduction

Researchers have proposed many approaches based on several techniques: information retrieval [5], events [13], hypertext [57, 64], scenarios [21], and rules [67], to recover traces among software artifacts. These proposed approaches use mainly three inputs for traceability recovery (TR): source documents, target documents, and experts' opinion. To the best of our knowledge, all the proposed traceability recovery approaches (TRA) have low recall and precision.

Our main claim in this chapter is that improving traceability recovery approaches only in themselves cannot help in improving precision and recall; we must also control the factors that impact the inputs of these approaches. To support this claim, we report in Table 1 the precision and recall values of some TRA described in the literature based on the following techniques: Vector Space Model (VSM), Latent Semantic Indexing (LSI), Rule-based, and Jensen-Shannon similarity (JS). It shows that, depending on the data sets, precision values vary from 0.9% to 95.9% and recall values vary from 3% to 99.8%.

Thus, Table 1 sustains our main claim by showing that different approaches using the same techniques report precision/recall values that vary a lot across data sets. For example, Sundaram *et al.* [68] achieved 1.5% to 7.9% precision with VSM whereas Abadi *et al.* [1] achieved 50% to 80% precision with the same techniques. Both groups of researchers used simple VSM to obtain their results and, therefore, factors other than the technique, VSM, are causing the variations in precision and recall.

Abadi *et al.* [1] and Sundaram *et al.* [68] used three different inputs: (1) source documents, (2) target documents, and (3) experts' opinion, who manually created oracles to calculate precision and recall and vetted the automatically-created links. Our main claim is that it is the variation in the three different inputs that mainly caused the observed variations in precision and recall.

Some researchers, *e.g.,* [3, 39, 2], have mentioned factors that impact TRA inputs. However, these factors and their impact on TRA have not received enough attention so far. Even given a TRA uses a technique that can return links with high precision and recall, if its inputs have poor quality, then this approach will produce

poor links. Thus, it is important to survey the factors impacting TRA inputs and report metric/tools to measure these factors and precautions to control them.

Typical Problematic Scenario. To understand how some factors impact TRA inputs further, let us consider a scenario where a project manager receives a verification and validation task. To complete her task, she needs up-to-date traceability links between requirements and source code. She uses a TRA that produces results with high precision and recall. She collects TRA inputs, such as experts' opinion, requirement specification document (RSD), and source code and asks the help of the best available resources, *i.e.,* a senior developer of the company with 10 years of Java and C++ programming experience. The senior developer can understand source code written in different programming languages; the updated requirements and latest source code provide traces that represent the actual system.

Let us now further assume that source code is in Perl, source code identifiers' quality is poor, a non-professional person wrote/updated the RSD. Then, it is likely that the expert would miss some links and retains erroneous links, because Perl has a different syntax and structure than Java and C++. The non-professional person would have probably written vague and ambiguous requirements in RSD that creates confusion while verifying links. The developers have used meaningless abbreviations for identifiers, thus complexifying the program comprehension activity. Therefore, the automatically-generated traceability links would be numerous and the expert would get frustrated and tired while verifying each and every one of them.

This scenario highlights the importance of TRA inputs in the TR process and of their analysis to help researchers and practitioners understand the outputs of TRAs.

Objectives and Overall Methodology. Given the importance of TRA inputs, project managers need guidelines to analyse TRA inputs and their impacting factors as well as metrics/tools to measure/improve the TRA inputs quality and preventive measures that must be taken to control the inputs quality. Thus, we define four objectives for this chapter. Objective 1 is to define and document the factors impacting TRA inputs. Objective 2 is to report metrics/tools to measure/improve the quality of the inputs by acting on the factors. Objective 3 is to provide preventive measures to control the factors. Objective 4 is to illustrate our main claim empirically using one of the identified factors: experts' programming knowledge.

To achieve our objectives, we follow the methodology depicted in Figure 1. In Step 1, we use our own traceability expertise [4, 5, 36] to define preliminary factors that, to the best of our knowledge, could impact TRA inputs. In Step 2, we perform an *incremental literature review* (ILR), using these first factors as seeds, to identify and define all the factors impacting TRA inputs and find evidence of their impact. We analyse experimental results reported in the literature that provide metrics/tools to measure/improve the quality of TRA inputs. In addition, we also identify and report precautions for the identified factors found in the literature. In Step 3, we document and report all the gathered data using a consistent template. The output of this step is reported in Section 3, which thus achieves our Objectives 1, 2 and

Fig. 1 Main objectives of the chapter and methodology to achieve these objectives

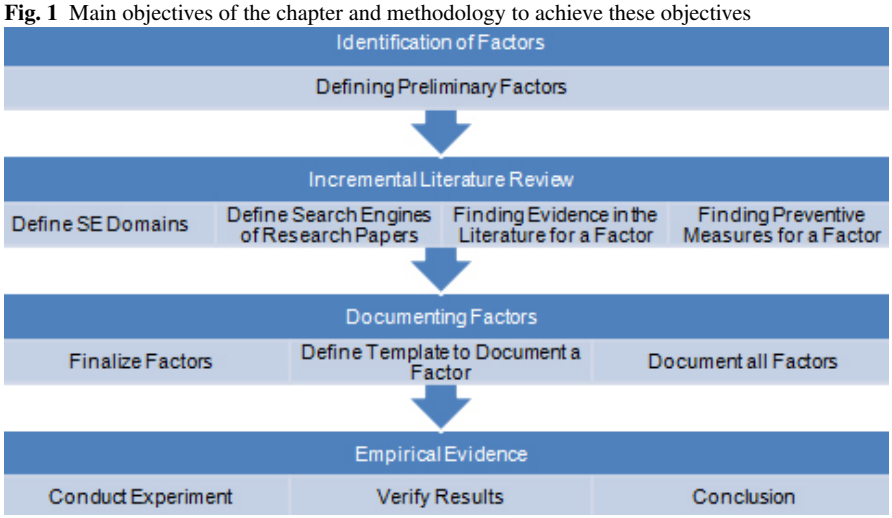


Table 2 Compulsory and Complementary inputs of TRAs

Approach	Compulsory Inputs	Complementary Inputs
Scenario-based	Requirements, source code	Hypothesized traces
		Execution traces (scenarios)
Rule-based	RSD, UCD	Requirement-to-object-model traceability rule Analysis object model Inter-requirement traceability rule
Event-based	Requirements, UML artifacts, and test cases	
Hypertext-based	Requirements, source code	Conformance analysis,
IR-based	Requirements, UML artifacts, and test cases	Thesaurus, temporal information (SVN, Bug reports, mailing lists) System dynamic information Experts' feedback

3. In Step 4, we perform an experiment on one factor that impact experts' opinion: experts' programming knowledge. Our empirical findings support our claim that factors impacting TRA inputs cause low precision and recall in state-of-the-art TRAs. This last step helps us achieving our Objective 4.

Assumption, Limitations, and Organisation. Table 2 shows the compulsory and complementary inputs for various TRAs gathered from the literature. These approaches considered, as compulsory inputs, requirements, use-cases, and UML artifacts as source documents and source code and test-cases as target documents. They also used complementary inputs as well that may vary for every TRA and have impact on the TRAs results. In the following, we only concentrate on compulsory inputs, because they are the same for all TRAs.

Table 3 Template to document TRA inputs, factors, and preventive measures

Attributes	**Descriptions**
TRA Input	Brief introduction to the TRA input
Factor Name	Name of the factor impacting the TRA input
Definition	Definition of the factor
Scenario	A scenario illustrating the impact of the factor
Literature Review	Literature evidence of the impact of the factor
Preventive Measures	Metrics, tools, and precautions to measure and control the factor

Thus, in this chapter and without loss of generality, we consider requirements as source documents and source code as target documents, to make it easier to describe the factors' impact on TRA inputs. This choice does not change the fact that these factors will impact any TRA inputs, if experts are recovering tractability links among requirements, between test cases and requirements, between scenarios and source code, and so on. For example, if experts are recovering traceability links among requirements then requirements would be both source and target documents, in this kind of situations same factors will impact source and target documents that impact requirements.

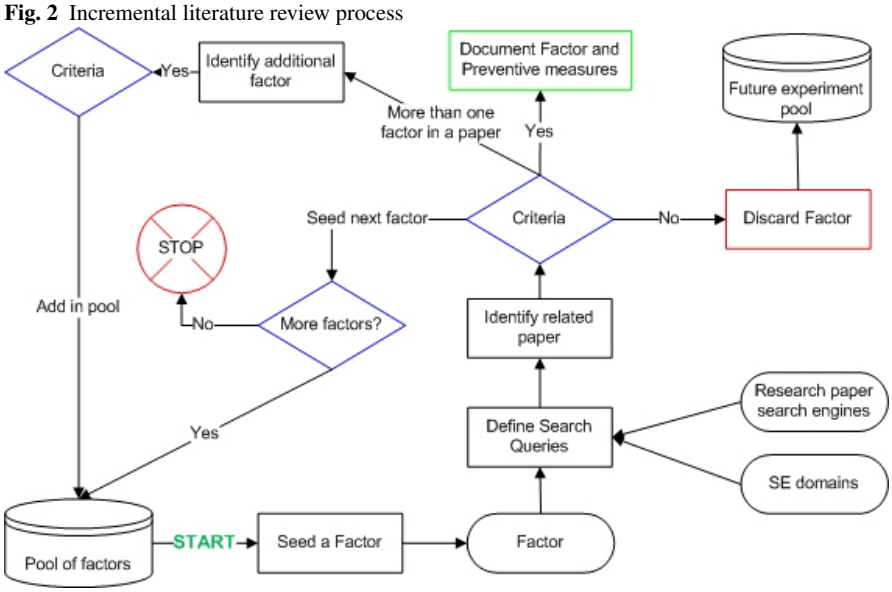
We do not report a systematic literature review but choose to rather perform an incremental literature review for reason of form and content. A systematic literature review would have required more space than available to report on all the papers related to the identified factors. Moreover, a systematic literature would have also required a set of predefined factors impacting TRA inputs and of formal criteria to assess these factors, both agreed-upon by the community [46]. Such factors and criteria, to the best of our knowledge, are not yet available in the literature and, thus, our incremental literature review of more than 60 papers is a first step towards identifying such factors and criteria.

The rest of the chapter is organised as follows: Section 2 describes our incremental literature review and summarises the retained factors that impact TRA inputs. Section 3 documents the factors, metrics/tools, and precautions for the retained factors. Section 4 describes our empirical study of the impact of experts' programming language knowledge on TRAs, its results, and threats to its validity. Section 5 discusses the findings in this chapter. Finally, Section 6 concludes with future work.

2 Identification of Factors and Preventive Measures

In the following, we first define an incremental literature review (ILR). Second, we perform a first ILR to identify and retain important factors according to our criteria. Third, we used these factors as input to a second ILR to identify preventive measures. Fourth, we document in Section 3 all the identified factors and preventive measures to measure/improve the quality of TRA inputs using a consistent template, described in Table 3.

Fig. 2 Incremental literature review process



2.1 Incremental Literature Review

We define an *incremental literature review* (ILR) to find factors impacting TRA inputs and evidence supporting their impact as well as preventive measures (metrics, tools, and precautions). Figure 2 shows the process that we followed in our ILR.

An ILR is a recursive process. It starts from a pool of eleven possible factors: Ambiguous Requirement, Vague Requirement, Conflicting Requirement, Identifiers' Quality, Domain Knowledge, Programming knowledge, Document Type, Document Language, Work Environment, Project Size, and Dead Code. We provide the definitions of seven of these factors in Section 4. We seed one factor to find evidence in the literature of its relevance and identify related research papers through queries in six software engineering sub-domains: information retrieval, program comprehension, requirements engineering, reverse engineering, software artifact traceability, and software maintenance. We use the same list of search engines for research papers for both ILRs, *i.e.,* IEEExplore[1], ACM Digital Library[2], Springer[3], and Google Scholar[4]. We verify if the identified paper provides evidence for the current factor or not. If the paper discusses more than one factor, we verify if the other factors also impact TRA inputs and, if they do, we add them to the pool for the next iteration.

[1] http://ieeexplore.ieee.org/Xplore/guesthome.jsp

[2] http://portal.acm.org

[3] http://www.springer.com

[4] http://scholar.google.ca

We use two sets of criteria to retain or put aside a factor. In the first iteration of our ILR, we put aside a factor from our study when we cannot find any evidence in the literature for that specific factor and–or when we can find only one paper that is not cited more than one time. In the second iteration of our ILR, we keep all the factors, even though we may not find papers describing related metrics/tools and–or precautions to highlight future research directions in Section 5.

We review and apply a set of criteria on the identified papers to answer two questions: (1) does any paper support the seeded factor? and (2) do the identified papers mention factors not already in the pool? From decision (1), we document or put aside the seeded factor for future experiments on their impact. From decision (2), we add to the pool of factors any missing factor. The process then iterates until there are no more factors to process. In the following two sub-sections, we perform and report the results of the two ILRs.

2.2 Identification of Factors

We listed (recall Section 2.1) of eleven factors that, to the best of our knowledge, could impact TRA inputs. We identified these factors based on our own traceability recovery expertise [4, 5, 36] and past professional experiences performing traceability recovery with private companies.

We performed an ILR for all the eleven factors to find out evidence that these factors impact TRA inputs. We seeded each factor in our ILR process to discover evidence in the literature supporting that the factor impact some TRA inputs. We defined search queries and looked for the papers in the chosen sub-domains using the chosen search engines. For example, we used the query *"identifiers quality"* in Google Scholar to identify the paper "What's in a Name? A Study of Identifiers" [49] supporting the factor "Identifiers' Quality".

After performing this first ILR, we could not find any research papers that clearly state that *Document Type*, *Document language*, *Work Environment*, *Project size*, and *Dead Code* and impact TRA inputs. Therefore, following our criteria, we remove these five factors from our study. Interestingly, we found one more factor during our ILR, *i.e.,* Granularity Level, which impacts TRA inputs as well as the overall economical aspect of traceability. We included this newly-found factor in our identified factors list. Figure 3 presents the final seven $(11-5+1=7)$ factors.

Figure 3 shows the three TRA inputs and the seven factors impacting these inputs. Rectangles represent TRA inputs and arrows represent the factors that impact these inputs. The last rounded rectangle represents the consequences of the factors on the TR process: incorrect/missed traceability links. We document each factor in Section 3.

Fig. 3 Inputs of traceability approach and impacting factors

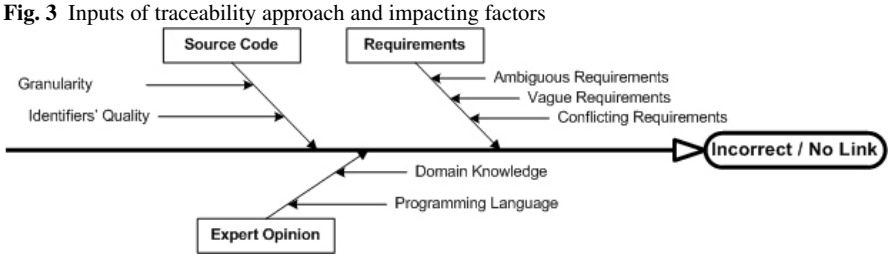
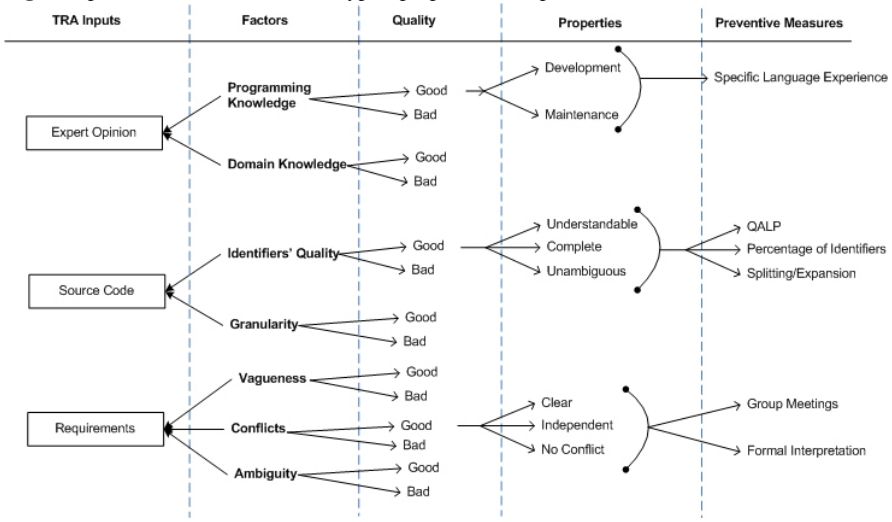


Fig. 4 Inputs of TRAs; factors, their types, properties, and preventive measures



2.3 Identification of Preventive Measures

Each factor can impact the TRA input negatively, yielding low precision and–or recall. We wanted to identify *metrics/tools* that can measure/improve the quality of TRA inputs. We associated some positive properties with each factor. For example, for Identifiers' Quality, identifiers must be understandable, complete, and unambiguous. Then, we searched for metrics/tools that are useful to measure/improve the quality of TRA inputs and factor with respect to their properties.

We seeded each retained factor in the ILR process. We followed the same steps as in the previous ILR. We analysed the *metrics/tools* reported in the literature for the measurement/improvement of the identified factors. We combined literature review and our own expertise to describe precautions for the factors. We discuss all the identified factors detail with their preventive *metrics/tools* in Section 3.

Figure 4 summarises the output of this ILR. It shows, for each factor, its five main characteristics: input name; factor name, type, property; and, preventive measures (metrics, tools, and precautions). For example, Source code is a TRA input and Identifiers' Quality impact source code. Good quality identifiers must be understandable, complete, and unambiguous. To obtain these properties, expert may use *splitting/expansion* [56] approach to split identifiers such as `cmdpntr` into `cmd pntr` and then expand the resulting words into `command pointer`. The results of the *splitting/expansion* approach have all the above-mentioned properties of good identifiers. Now, let us assume that an expert is using an IR-based approach to recover traceability links between requirements and source code, the split and expanded identifiers would link to `command pointer`-related requirements more likely than the `cmdpntr` identifier would.

3 Factors Impacting the Inputs of TRAs

Researchers have proposed various TRA, *e.g.,* [5, 13, 21, 64, 67]. To the best of our knowledge, all of these approaches have low recall and precision. Recall is defined as the *number of relevant documents retrieved* divided by the *total number of relevant documents*:

$$Recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|}$$

while precision is defined as the *number of relevant documents retrieved* divided by the *total number of retrieved documents*:

$$Precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|}$$

The low precision and recall of the retrieved links impact the usefulness of the TRAs. Low precision requires experts to deal with numerous spurious traceability links while low recall casts doubt in the experts' minds about missing links and requires them to analyse by hand artifacts to possibly identify these missing links.

Researchers proposed various methods [25, 26, 48] to improve the precision and recall of TRAs. However, to the best of our knowledge, there has been little work [3, 39, 2] on the factors that impact TRA inputs. We now document three types of TRA inputs (requirements, source code, and experts' opinions), factors impacting these inputs, metrics/tools to measure/improve the input quality, and precautions to control the factors, using the template shown in Table 3.

3.1 Requirements

The precise capture, understanding, and representation of requirements is a crucial step in the development of effective and usable information systems [27]. Requirements are often error-prone due to misinterpretation of natural languages [24]. For example, if a requirement is incomplete, such as *change time*, it may trace to session time, patient wait-time, or system time; it would be difficult for an expert to verify its corresponding traceability links. In general, completeness and correctness depend on several factors. Hayes [40] reports 13 factors, including ambiguous and non-verifiable requirements. In the following, we only report three factors for which, using our ILR, we could find definitions, experimental results, and precautions.

3.1.1 Ambiguous Requirements

***Definition*:** Ambiguous requirement are requirements of which two different experts may have different interpretation [11].

***Scenario*:** Ambiguous requirements may result into different interpretation and implementation. They lead to perplexity and waste of effort during their understanding. They also impact the TRAs by leading to the creation of ambiguous links that are complicated to verify. For example, in the requirement *"each new user shall be part of a group"*, the concept of *group* could be ambiguous and an expert could interpret this group to manage access privileges, whereas another expert may interpret it as a group of common, shared interests.

***Literature Review*:** Ambiguity has long been pictured as one of the worst enemy of experts writing requirements, especially with reference to ambiguity in natural language requirements [25]. Zisman *et al.* [72] mentioned that the main shortcoming of TRAs is their inability to automatically identify and maintain traceability relations involving natural-language artefacts with ambiguous meanings. Hayes *et al.* [37] showed in their paper that senior analysts at Science Applications International Corporation missed 17 links during a manual traceability link recovery activity. The authors' observations on the missing links was: (1) it was difficult to do some of the tracing because the documents/requirements were incomplete, ambiguous and (2) unknown acronyms hindered the trace recovery process.

Haiduc and Marcus [31] studied several open-source systems and found that about 40% of the domain terms are being used in the source code by developers. If the domain terms are ambiguous, it will also impact source code as well.

Hayes [40] presented a methodology for requirement-based fault analysis and its application to NASA software projects. She examined requirements faults for the International Space Station (ISS) software systems. She showed that 6.1% of the faults were caused by ambiguity in the requirements of the ISS.

***Preventive Measures*:** Some approaches [11, 28, 43] have been proposed by researchers to identify ambiguity in and remove them from requirements. Gleich *et al.* [28] presented a tool to detect ambiguities and to explain the sources

of these ambiguities. They claimed that their ambiguity-detection tool yields a significant improvement in time and cost and in quality in industrial contexts. Kamsties *et al.* [43] presented an inspection technique for detecting ambiguities in informal requirements. Their results showed that inspection techniques yield better results than formal methods in term of the number of identified ambiguous requirements.

3.1.2 Vague Requirements

***Definition*:** Vague requirements are imprecise natural language statements. If the statements of the requirements fail to draw an image or bring an understanding of what is desired, then they are vague because difficult to interpret correctly [42].

***Scenario*:** For example, the requirement "*maintaining patients' records shall be good*" is vague. The word "good" is not defined. An expert cannot trace the implementation of "good" patient records into any source code.

***Literature Review*:** Erik *et al.* [44] conducted case studies with ten different small and medium enterprises (SMEs). They mentioned that SMEs do not document requirements properly, which cause problems such as (1) requirements are too vague or prosaic to be testable, (2) requirements are not traceable, and (3) the domain knowledge implicitly contained in requirements makes the requirements difficult to understand by developers.

Kasser [45] stated that vague requirements cause expensive cost and delay in project schedule. They mentioned that vague requirements are unverifiable and contain multiple requirements in a single paragraph, which complicate the traceability of tests to requirements.

Ghazarian [26] showed that 57.5% of bug reports are due to incorrect implementations of requirements in the source code. Vague requirements cause this kind of reports.

Hall *et al.* [32] studied the problems experienced by 12 software companies in their requirement process and showed that 48% of their problems stem from requirements and that vague requirements cause 25% of these problems.

***Preventive Measures*:** Kasser [45] presented a tool, FRED, to detect vague requirements and allow an expert to remove the vagueness from the requirements. FRED also helps to make requirements traceable by splitting two combined requirements.

Lee *et al.* [51] proposed the Requirements Trade-off Analysis technique to formalise vague requirements. They analysed trade-off among vague requirements by identifying the relationship between requirements, which could be either conflicting, irrelevant, cooperative, counterbalance, or independent. Fabbrini *et al.* [24] proposed a tool, QuARS (Quality Analyzer of Requirement Specification), based on their natural-language quality model to detect vague requirements.

3.1.3 Conflicting Requirements

***Definition*:** Conflicting requirements are requirements that are incompatible in a same or different artifacts [40].

***Scenario*:** During the requirement elicitation process, each stake-holder gives her wish list without considering conflicts with other stake-holders' requirements [42]. For example, one stake-holder could ask that the system shall allow giving bonuses after six months while another stake-holder could ask that the system gives bonuses every three months. Such conflict could result into two separate implementations of the requirements that may then conflict and must be maintained separately. It will also create problems for the experts verifying whether the system allows bonuses.

***Literature Review*:** It is risky to ignore or stifle conflicting requirements because they may have serious negative consequences on the software development process [30]. Many researchers have highlighted the significance of identifying and analysing conflicting requirements for the success of system development [30, 35, 34, 42].

Egyed *et al.* [22, 21] conducted requirements traceability studies on a video-on-demand system. They found that some requirements have dependencies with other requirements and that these dependencies cause conflicts. For example, in order to start playing a movie, one needs to load the textual information about the movie, which is allowed to take up to three seconds while 1 second is the required maximum duration before starting playing a movie. Egyed *et al.* recommended that conflicts and dependencies be removed before performing traceability tasks.

Hayes [40] divided conflicting requirements into internal and external conflicts [40]. She showed that 4.7% of the faults in the ISS software systems are due to conflicting requirements.

***Preventive Measures*:** Stake-holders must discuss and resolve conflicting requirements [30, 42, 35]. They can negotiate the conflicting requirements. Egyed [21, 19] proposed a tool-supported approach, Trace Analyser, to analyse dependency among requirements and detect conflicts. Trace Analyser cannot automatically derive conflicts but, by finding all possible requirement dependencies, it makes it easier to identify potential inconsistencies and conflicts.

Hausmann *et al.* [34] presented a formal interpretation of use-case models, which is based on concepts from the theory of graph transformation. Use-case models allow to define precisely the notions of conflict and dependency between functional requirements. Then, use-case models can be statically analysed to identify conflicts and dependencies, which can then be communicated to the stake-holders by annotating the model. They also provided an implementation of the static analysis within a graph transformation tool.

3.2 Source Code

Source code is a common input for of traceability approaches [5, 58]. The quality of the results of a TRA highly depends on the quality of the source code [3]. For example, if a developer uses meaningless abbreviations for identifiers, thus use causes low similarity between requirements and source code [49, 50] and creates ambiguity for an expert when verifying recovered links. Moreover, some techniques, such as information-retrieval techniques [5, 37, 52, 54], require high-textual similarity to recover traceability links.

Developers normally use identifiers [9] that are easy to remember. However, these identifiers possibly do not represent concepts in the source code and–or system domain. For example, developers usually use i, m, n, and k as variable names for integer values, but these do not represent any concept. Such identifiers can result into low textual similarity and poor links. Appropriate use of identifiers does not only help to improve TR, it also helps in improving the overall software quality [48].

Developers also often mix different concepts in the same classes and implement as much functionality as possible in a single class under time pressure to implement as quickly as possible new functionalities [59, 60]. This "design choice" or, rater, lack thereof, creates overlapped links [22] that are difficult for experts to sort and verify manually.

3.2.1 Granularity Level

***Definition*:** Level of detail considered in the TR. Granularity is generally divided into three levels: coarse, middle, and fine-grained. As the level of granularity increases, a TRA would provide more detailed and numerous links.

***Scenario*:** Let us assume that a developer implements different concepts in the functions of one object-oriented class. Typically, a developer creates one `Patient` class and implements all patient-related concepts in that class in the form of methods, such as adding walk-in patient, adding emergency patient, and so on. Let us now assume that an expert is recovering links between the requirements and classes of this system. Then, several requirements may link to that one `Patient` class, which would impede the experts' verification of the links if the requirements are at the class level.

***Literature Review*:** Egyed *et al.* [23] showed that tracing requirements to method level requires $3-6$ times more effort than tracing requirements to classes. They showed that links at the method-level have no advantage over links at the class-level in terms of quality.

Bianchi [6] *et al.* conducted an exploratory case study to evaluate the relationship between the granularity of a traceability model and the effectiveness of the maintenance process. Their case-study results showed that fine-grained traceability requires greater effort to satisfy maintenance requests but also provides better accuracy. Therefore, experts must trade effort for accuracy.

It is equally important to consider the return on investment (ROI) [23] of traceability [20] while choosing a granularity level. Egyed *et al.* [20] evaluated the ROI of tracing at lower levels of granularity. They measured the ROI by the effort needed to recover the links against the value returned through tracing at different levels of precision. Their case study showed that a tenfold increase in cost/effort only produces twofold improvement in precision.

***Preventive Measures*:** It is important to choose the "right" granularity level [6, 12] before starting a TR process. If developers used switches to handle different requirements then it is important to choose a finer-grain granularity. If the ROI is not high then the experts may perform refactoring tasks to separate different implementations of requirements at class level to reduce the number of traceability links. Developers should implement different requirements in different classes, if they are working with an object-oriented programming languages, or in different functions and modules, if they are working with a procedural programming language.

3.2.2 Identifiers' Quality

***Definition*:** An identifier is the name of a token in the source code. Software quality depends on identifiers' quality [9] because the majority of the source code of a software system consists of identifiers [16].

***Scenario*:** If a developer used meaningless abbreviations to name identifiers, it will create problem for any automated or manual TR process. For example, if the developer has named a method "*cd*" in a *file management system*, then an expert verifying a traceability link for the *create directory* requirement, would not be able to easily distinguish between *change directory* and *create directory*. The expert must consequently have to read the whole source code to keep or reject the traceability links.

***Literature Review*:** Several studies showed that poor identifiers' quality impacts TR [53, 55, 15]. De Lucia *et al.* [53] used traceability to identify poor quality identifiers. They used a IR-based traceability approach to build links between source code and high-level documents. Their approach highlights the identifiers whose understandability is decreasing due to continuous software maintenance and evolution. Their studies showed that using meaningless identifiers could result into poor quality traces.

Butler *et al.* [9] analysed the impact of naming conventions on maintenance effort, *i.e.,* on code quality. They evaluated the quality of identifiers in eight open-source Java libraries using twelve naming conventions. They showed that a statistically-significant relation exists between identifiers and software quality.

Takang *et al.* [70] compared abbreviated identifiers with full-word identifiers and uncommented code with commented code and empirically analysed the role played by identifiers and comments on source code understandability. They showed that (1) commented systems are more understandable than non-

commented systems and, similarly, that (2) systems containing full-word identifiers are more understandable than those with abbreviated identifiers.
De Lucia etal [55] stored poor links during traceability recovery experiments to analyse them. They found that poor links helped to identify quality problems in the textual descriptions of the traced artifacts; mainly a poor description of the artifacts. The expert used these poor links to improve the textual description of the artifacts. As a result of these changes, over 60% of the poor links highlighted by the tool improved with a similarity value above the quality threshold at the end of the project.

***Preventive Measures*:** Improving identifiers' quality yields an increase in precision and recall of TRAs. For example, if two concepts are merged in one identifier, it is important to split this identifier to avoid ambiguity among identifiers. Researchers have proposed different approaches [7, 8, 56] to improve the identifiers' quality. Madani *et al.* [56] proposed a speech recognition-based approach to split identifiers and expand them; their approach out-performs the Camel Case splitter.
QALP [8] metrics calculate scores between source code identifiers' and comments. High scores highlight a strong relationship between source code and comments. QALP helps to identify any ambiguity among different identifiers and comments. Lawrie *et al.* [7] proposed an approach based on the percentage of identifiers that violate syntactic conciseness and consistency rules. Their approach helps to avoid confusing identifiers.

3.3 Experts' Opinion

The field of human factors research is large and diverse. As of today, no large-scale study involving human experts has been conducted in TR [14]. Different studies show the importance of experts' opinion in TR [14, 17, 18, 66]. Ghazarian *et al.* [26] showed that developers cause 82% of the problems of missed implementation in some software systems. Hayes *et al.* [38] conducted a case study on experts' feedback. They asked three experts to perform some traceability tasks on three different data sets. They give the experts traceability links with low precision, with high recall, and with high precision and low recall. They showed that experts were not able to provide better results than the tool. They mentioned that there may be other factors, such as domain knowledge, impacting the experts' results.

Expert may analyse false positive links generated by tools but would need lots of efforts to create missing links by analysing the software artifacts manually. In addition, if an expert generates incorrect links, there are usually no second verification; therefore, it is important to analyse the human factors impacting TR. Below are some of the main factors that impact an expert's opinion.

3.3.1 Domain Knowledge

***Definition*:** Domain knowledge characterises an expert's understanding of the field in which the analysed software system is being developed.

***Scenario*:** Experts use their domain knowledge to query the software artifacts for specific concepts. For example, an expert, who does not have Web development experience and wants to search for a function that return all the variable values from a URL, may use keywords such as "*URL*", "*value*", "*get values from URL*", "*URL variable values*", and so on. Thus, the expert wastes effort, making inaccurate queries; whereas, with appropriate domain knowledge, she would simply use the query "*query string*" to search the relevant function.

***Literature Review*:** Hayes *et al.* [38] reported that several factors impact the quality of experts' opinion, including their domain knowledge.

Taira *et al.* [69] conducted an empirical study to identify the impact of domain knowledge when learning with the help of a search engine. Their results showed that confusion in Web surfing was caused by a lack of knowledge in the domain of interest. They observed that domain knowledge may assist an expert in avoiding being confused and in finding suitable Web pages.

Park and Black [62] performed an experiment to investigate the impact of domain knowledge on search activities. Their results showed that domain knowledge impacts the precision of search results.

***Preventive Measures*:** Domain knowledge improves the experts' opinion during traceability tasks. Thus, an expert must acquire adequate domain knowledge before exploring source code and other artifacts. An expert can obtain domain knowledge by using the software system [63]. An expert must also have enough time to learn and understand the overall functionality of the system to consequently be able to recover/verify traceability links adequately.

3.3.2 Programming Knowledge

***Definition*:** Programming Knowledge relates to an expert's ability to solve programming problems and write quality software in a particular programming language.

***Scenario*:** An expert in Java may not be able to understand Smalltalk source code adequately. Indeed, if an expert does not have Smalltalk programming knowledge, she may find concept in source code that she cannot readily understand or could misunderstand. For example, if an expert, with Java programming experience, queries some Smalltalk source code for "*add patient*", she may find the string *"add patient"* and think that it has something to do with the corresponding functionality while quoted strings in Smalltalk are comments.

***Literature Review*:** Studies [37, 38] showed that experts can recover false links and skip correct links during TR. However, to the best of our knowledge, these studies did not consider the experts' programming knowledge. It is quite possible that experts who vet the final traceability links have good programming knowl-

edge experience [66] but not of the specific language that the current system uses.

Chan [10] performed an empirical study with 100 undergraduate students to measure the effect of domain-specific knowledge and programming knowledge for software maintenance tasks. Their study showed that both programming and domain-specific knowledge have a significant impact on software maintenance productivity. They also discussed that hiring fresh graduates for maintenance tasks can increase the effort and cost.

Lau *et al.* [47] conducted an empirical study on 217 secondary students to measure the effects of gender and learning styles on computer programming performance. Their results showed that there is no significant effect of gender on programming performance, but academic ability had a differential effect on programming knowledge. Sequential learners [29] in general performed better than random learners [29].

***Preventive Measures*:** Domain knowledge is important to understand a system internal workings. Yet, we cannot ignore that general and specific programming knowledge also matter. It is important that the experts must be knowledgeable of the programming language that the analysed software system uses when performing TR. An expert with one-year Smalltalk programming experience can understand Smalltalk source code better than an expert who has ten years of Java experience, as discussed in Section 4.4. Expert must be selected based on specific programming language experience if possible.

4 Empirical Study for a Factor Impacting the Inputs of TRAs

Goal. We want to quantify the impact of one identified factor on TRA input. Quantifying one of the factor's impact on TRA inputs is one more step towards improving TR process. We select experts' programming knowledge because if other factors impact TRA to create wrong links or miss correct links, then expert could create/recover that links. However, if experts create wrong links or miss correct links then these links are likely to be so forever.

Study. We study whether experts without the programming knowledge of a system under analysis can perform different traceability tasks, such as creating links missed by TRAs and verifying links recovered by TRAs. We use Java and PHP as programming languages. We use 40 subjects from both industry and academia, divided in two *groups*; the *first group* with good Java knowledge and the *second group* with good PHP knowledge. We ask all the subjects to create and verify traceability links for Java and PHP systems. We measure the subjects' performance with: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentage of correct answers.

Results. Collected data shows that, in the first group, Java experts' programming knowledge positively impacted their results when they performed TR tasks for a Java system and negatively when they performed TR tasks on a PHP system. In the second group, for PHP experts and Java and PHP systems, collected data show the inverse results.

Relevance. Understanding the impact of the factors is important from the point of view of both researchers and practitioners. For researchers, our results bring further evidence to support our claim of the impact of the identified factors on TRA inputs. For practitioners, our results provide concrete evidence that they should pay attention to the identified factors to improve their TR process and use the reported preventive measures to handle these factors. Our results support our claim that it is also important to control TRA impacting factors to improve precision and recall.

4.1 Experimental Design

Our experiment uses two groups, the subjects in the *first group* have expertise in Java but not in PHP whereas those in the *second group* have expertise in PHP but not in Java. We use a within-subject design [65] in this experiment. An advantage of the within-subject experimental design is that confounding variables due to differences in subjects' skills are reduced [71].

4.1.1 Research Question

The goal of our experiment is to analyse how experts' programming knowledge supports or hinders experts' opinion during TR tasks. The experiment addresses the following research question: **RQ – Experts' Programming Knowledge**: *Will experts with specific programming language experience provide better traceability results than others?*

We try to reject the following null-hypothesis: *The presence or absence of experts' specific programming language knowledge has **no** statistically significant effect on average performance while performing requirement traceability tasks*.

4.1.2 Subjects Selection

The subjects are volunteers. Subjects have guaranteed anonymity and all data has been anonymised. We received the agreement from the Ethical Review Board of École Polytechnique de Montréal to perform and publish this study. The subjects

Table 4 Average precision and recall range of TRAs

	Industry	Academia	Industry		Academia	
	General Programming	General Programming	Java	Php	Java	Php
Group I	1.038461538	4.75	0.503846154	0.038461538	2.280769231	0.307692308
Group II	2.122142857	3.857142857	0.042857143	1.550714286	0.492857143	0.635714286

were aware that they were going to perform requirement traceability tasks, but do not know the particular experimental research question.

We recruited subjects from academia and industry to make sure that academic or industry experience has little impact on our experiment. There are 26 subjects from academia and 14 from industry. Industrial subjects have between 11 months to 5 years industrial experience whereas academic subjects are M.Sc. and Ph.D. students at École Polytechnique de Montréal (ÉPM). Industrial subjects are currently working in industry and academia subjects are currently enrolled at ÉPM. Table 4 shows the subjects programming experience statistics. We only consider a subject expert in Java or PHP, if she has more than four months experience in Java or PHP.

In the first group, the subjects have expertise in Java, Eclipse, basic domain knowledge of content management systems and medical systems, and no expertise in PHP, to qualify for the experiment whereas in the second group, they have expertise in PHP, Eclipse, basic domain knowledge of content management systems and medical systems and no expertise in Java.

4.1.3 Source Code Selection

We used several criteria to select the systems used in our experiment. First, we selected open-source software systems, so that other researchers can replicate our experiment. Second, we avoided small systems that do not represent systems handled by most developers. Finally, we conducted a pre-experiment survey about the subjects' known systems. We selected the systems that subjects did not know to avoid any learning bias. For the experiment, we used iTrust [5] and Joomla[6]. iTrust is developed in Java. It is an online medical record system with 19,604 KLOC, 526 classes, and 3,404 functions. Joomla is a content management system developed in PHP with 203 KLOC, 737 classes, and 4,834 methods.

4.1.4 Links, Tasks, and Questionnaires

The first author created traceability links manually between source code and requirements for Joomla and the second author verified these links to avoid bias. For iTrust, we used the links that iTrust developers provided us. The manually-created links help to evaluate subjects' answers. (One of the subject performed a pilot-study

[5] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php

[6] http://www.joomla.org

Table 5 Experimental questionnaire format

Category 1: *recover traceability links*	
Question 1 & 2	System shall allow ***this functionality***
Category 2: *Verify traceability links.*	
Question 3 (a,b,c)	System shall allow ***this functionality***, links to ***this class or method***.

to validate that the requirements used in the experiment are clear and simple. We excluded this subject and pilot-study from our final results.)

In any traceability task, an expert must verify traceability links created by a TRA and create new links that the TRA missed. We designed our questionnaire to address both these tasks. We asked two set of questions to the subjects, in two categories. In the first category, we asked subjects to create missing traceability links among requirements and source code. This category contains two questions for each system. We used vector space model (VSM) [5] to automatically create traceability links between requirements and source code. VSM provided true and false traceability links. In the second category, we asked subjects to verify requirement traceability links recovered by VSM as true or false. This category contains three questions for each system. In the second category, the first system contains 2 true and 1 false traceability links, whereas the second system contains 1 true and 2 false traceability links. Table 5 shows the experimental categories and questions, the text in bold is a placeholder that we replace by appropriate required behaviour of the systems.

For example, with Joomla, we replace *"this functionality"* in Question 1, Category 1, by *"update any article's contents"* and the question reads as: *system shall allow* ***updating any article's contents***. In Question 3(a), Category 2, we replace *"this functionality"* and *"class"* by *"administrator to add different sections in website"* and `administrator.components.com_sections.admin.sections.php` and the question reads: *"System shall allow* ***administrator to add different sections in website****"* links to `administrator.components.com_sections.admin.sections.php`.

4.2 Procedure

We divide the experiment into three steps. In the first step, subjects are explained the systems. We provide basic details of the systems on the answer sheets. In the second step, we ask the subjects to provide their general Java or PHP industrial and academic programming experience in years. To confirm the subjects' experience, we ask them the maximum source code size that they have developed in the past. We consider that a subject has expertise in a programming language, if she has more than four months experience and has written more than $5,000$ LOCs. We use four months because in academia a semester duration is four to six months and in industry it is considered as probation period. Therefore, considering a subject who is currently studying a programming language subject or industry subject who is

in probation period could bias the results. In the third step, we ask the subjects to recover and verify traceability links.

For each system, we ask subjects to spend adequate time to explore the code and perform their traceability tasks. We prepare each of the target system in an Eclipse Workspace. We provide the subjects with a timer, developed in Java to record the time that they take to answer a question. We ask subjects to start the timer when they begin looking for an answer and stop when they find the answer. We ask subjects not to start the timer when they are reading and understanding a question or writing an answer.

We measure the subjects' effort using the NASA Task Load Index (TLX) [33]. The TLX assesses the subjective workload of subjects. It is a multi-dimensional measure that provides an overall workload index based on a weighted average of ratings on six sub-scales, *i.e.,*, mental demands, physical demands, temporal demands, own performance, effort, and frustration. NASA provides a computer program to collect weights and ratings for the six sub-scales. We combine all workload factors to compute an average workload for each subject. To combine all workload factors, each rating is multiplied by the weight given to that rating by the subject. The sum of the weighted ratings for each task are divided by 15 to get the average workload [33].

4.3 Analysis Method

We perform the following analysis to answer our research question and attempt rejecting our null hypothesis. We use programming language knowledge as an independent variable whereas time, percentage of correct answers, and effort are dependent variables. We divide the total number of correct answers by the total number of questions to obtain an average of correct answers for each subject.

We use the Mann-Whitney test to compare the two sets of dependent variables and assess whether their difference is statistically significant. The two sets are the subjects' data that we collected when they answered traceability questions with or without specific programming expertise. For example, we compute the Mann-Whitney test to compare the set of average correct answers of Java experts with non-Java expert for the Java system. Mann-Whitney is a non-parametric test; therefore, it does not make any assumption about the distribution of the data.

We compute the Cohen's d impact size [65], which indicates the magnitude of the effect of a treatment on the dependent variables. The effect size is considered small for $0.2 < d < 0.5$, medium for $0.5 < d < 0.8$ and large for $d > 0.8$. It is defined as the difference between the means $(\mu_1 - \mu_2)$, divided by the pooled standard deviation $\sqrt{\left(\sigma_1^2 - \sigma_2^2\right)/2}$ of both variables: $d = (\mu_1 - \mu_2)/\sigma$.

Table 6 Experiment Result's Statistics

Factor: Expert Programming Knowledge					
Systems	**Knowledge**	**# of Subjects**	**Correct Answers**	**Times**	**Efforts**
iTrust (Java)	Good	26	69.23	186.04s	30.12
Joomla (PHP)	Bad	26	30.00	379.00s	60.02
iTrust (Java)	Bad	14	44.29	205.93s	39.33
Joomla (PHP)	Good	14	80.00	87.57s	49.07

Table 7 *M*ann Whitney p-values, precision, recall, and Cohen's d effect size for each experiment

	Time		**Answers**		**Efforts**	
	M.-W. p	**Cohen** d	**M.-W.** p	**Cohen** d	**M.-W.** p	**Cohen** d
Group I	0.000009	2.38	0.000015	2.40	0.000001	2.79
Group II	0.001094	1.89	0.001453	2.43	0.003052	1.62

4.4 Experimental Results

After collecting the answer sheets, we compared subjects' responses with the predetermined correct answers to compute the average correct answers for each subject. Table 6 shows the statistics of the results, all the results' values are average values. For example, in the first group with Java expertise, the subjects took an average of 186.04 seconds to answer a question for the Java system while they took on average 379 seconds to answer questions for the PHP system. Table 7 shows the p-values and Cohen's d values calculated by comparing the differences between the data collected for each experiment.

There is statistical significant evidence to reject the null hypothesis. Table 7 shows that the p-values are below the standard significant value, $\alpha = 0.05$. Moreover, the Cohen's d values are also high (> 0.8). Subjects with expertise in Java were able to create/verify more correct links in less time and with less effort than the subjects who did not have expertise in Java for the Java system and vice-versa for the PHP experts and the PHP system. We also find some interesting observations. In both groups, there are 3 subjects who are good in both PHP and Java. They performed better than other subjects on both systems by spending less effort and time to find the correct answers.

Thus, we answer the RQ as follows: programming knowledge does impact experts' opinion. It is important for an expert to have good knowledge of the specific programming language in which the system under analysis is written.

4.5 Threats to Validity

Several threats potentially impact the validity of our experimental results. We discuss below these threats and how we alleviate or accept them.

Construct validity: The construct validity concerns the relation between theory and observations. In this experiment, it could be due to measurement errors. The average correct traceability links created and time spent by a subject, are the main measure in our study. As the correct answers are predetermined before conducting the experiment, measuring individual subject's performance is simply a matter of comparing each subject's answers with the expected correct answers.

Internal validity: The internal validity of a study is the extent to which a treatment effects change in the dependent variable. There can be learning threat in our experiment. We used two different systems and different kinds of requirements' links to avoid this learning threat. We give subjects an opportunity to ask any questions that they may have about the material. While answering their question, we were careful not to reveal any information that could help them to find the correct answers. We only explained what was already available in the training material. We also instruct subjects not to discuss the experiment among themselves. The source code of both systems was not same and we used two groups with different expertises to avoid source code size effect on our results.

External validity: The external validity of a study relates to the extent to which we can generalize the results of our studies. To avoid any external validity threat, we engaged subjects from academic and industry to help generalising our findings to both contexts. Moreover, we performed our study with 40 (26 for first group and 14 for second) and we used two different systems in different languages, Java and PHP. Yet, we cannot claim to generalise our results to other programming languages.

There were only five links (two links to recover and three links to verify) in our experiment, while the traceability links that experts recover or verify in practice are numerous. One major reason to use few traceability links was experimental control. Table 7 shows that the results are significant and that, moreover, the magnitude of the observed effects is large and thus cannot be ignored. Our preliminary study support the claim that it is important to control the factors that impact traceability approaches inputs.

Conclusion validity: Conclusion validity threats deals with the relation between the treatment and the outcome. We paid attention not to violate assumptions made by statistical tests. Therefore, we used a non-parametric test that does not make any assumptions about the distribution of data.

5 Discussions

We now discuss four questions related to the discussed factors and our methodology.

How much does controlling all the factors increase an experts' workload. Using poor quality TRA inputs will result in large number of false positive and missing links. It could be easier for an expert to improve TRA inputs quality than to manually recover missing links and verify large amount of false positive links. Improving TRA inputs may also help during program comprehension, maintenance, and reuse.

In future work, we will perform empirical studies to see how much time and effort can be saved if we control these factors' effect on TRA inputs.

Is it possible to control all the factors. It might not be possible for an expert to control all these factors, but controlling the maximum possible number of factors could still yield better results than using poor quality inputs. Future work includes developing and assessing the cost model to compute the ROI of controlling the various factors and combination thereof.

What are the most critical factors. We have not performed a systematic literature review due to space limitations. However, we performed an ILR and found more than 60 papers related to our study. The provided list of factors is a starting point towards traceability improvement by controlling these factors. We will perform in future work a systematic literature review to attempt identifying all factors and their impact on TRA inputs.

Are there more factors then these seven identified factors. We excluded five factors from our study: document type, document language, work environment, project size, and dead code because we could not find significant evidence in the literature that these factors impact TRA inputs. Yet, these excluded factors may impact TRA inputs. For example, if a TRA takes as input some source code and requirement documents and computes their textual similarity, it is possible that many requirements would link to dead code and an expert would have to verify these links even though they are useless. We add these excluded factors in our list of future experiments to quantify their impact on TRA inputs.

6 Conclusion and Future Work

It is important to develop new and improve existing traceability approaches, but it is also important to gain a better understanding of and support for the factors that impact TRA inputs. We claimed in this chapter that some factors impact TRA inputs, in particular, source code, requirements, and experts' opinion.

We defined a methodology to identify in the literature factors impacting TRA inputs as well as their definitions and associated preventive measures. Our methodology is based on two incremental literature reviews (ILRs) to identify critical factors that impact TRA inputs and tools/metrics to measure/improve the quality of the TRA inputs. We also used the ILRs to collect precautions to control the effect of the factors. We documented seven factors using a consistent template and rejected five factors for which we could not find enough supporting evidence.

To empirically support our claim, we conducted an empirical study to measure the experts' programming knowledge impact on experts' opinion. We showed that a group of Java experts could not perform well traceability-related tasks on a PHP

system, while PHP experts could, and vice-versa for PHP and Java experts on a Java systems. These results support our claim that expert's programming language knowledge impacts TRA inputs. Thus, the expert must be knowledgeable about the programming language(s) that the system under analysis uses.

In future work, we will perform a systematic literature review to identify more factors and their effect on TRA inputs. We will perform more experiments on all other remaining factors to assess their impact on TRA inputs. We will provide a priority list for project managers so that they can find which factors can impact their inputs more. We also want to provide a TR process that automatically handles potential factors impacting any TRA inputs.

References

1. A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 103 –112, june 2008.
2. N. Ali. Trustrace: Improving automated trace retrieval through resource trust analysis. In *ICPC '11: Proceedings of the International Conference on Program Comprehension (ICPC'11)*, page 4, Washington, DC, USA, 2011. IEEE Computer Society.
3. G. Antoniol. *Recovery of traceability links in software artifacts and systems*. PhD thesis, Montreal, Canada, 2003.
4. G. Antoniol, J.H. Hayes, Y.-G. Guéhéneuc, and M. di Penta. Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 28 2008.
5. Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation, October 2002.
6. A. Bianchi, A.R. Fasolino, and G. Visaggio. An exploratory case study of the maintenance effectiveness of traceability models. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 149 –158. IEEE Computer Society, 2000.
7. D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Software Fault Prediction using Language Processing. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 99–110. IEEE, 2007.
8. D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software*, 82(11):1793–1803, 2009.
9. S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Relating identifier naming flaws and code quality: An empirical study. volume 0, pages 31–35, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
10. T. Chan. Impact of programming and application-specific knowledge on maintenance effort: A hazard rate model. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 47–56. IEEE, 2008.
11. F. Chantree, B. Nuseibeh, A. de Roeck, and A. Willis. Identifying nocuous ambiguities in natural language requirements. In *Requirements Engineering, 14th IEEE International Conference*, pages 59 –68, 2006.
12. J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *Computer*, 40:27–35, 2007.
13. J. Cleland-Huang, C. K. Chang, and M. Christensen. Event-based traceability for managing evolutionary change. *IEEE Transaction Software Engineering*, 29(9):796–810, 2003.
14. J. H. Hayes D. Cuddeback, A. Dekhtyar. Automated requirements traceability: the study of human analysts. Los Alamitos, CA, USA, 2010. IEEE Computer Society.

15. A De Lucia, R. Oliveto, and G. Tortora. Assessing ir-based traceability recovery tools through controlled experiments. *Empirical Software Engineering*, 14:57–92, February 2009.
16. F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14:261–282, 2006.
17. A. Dekhtyar, J.H. Hayes, and J. Larsen. Make the most of your time: How should the analyst work with automated traceability tools? In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, pages 4 –4, May 2007.
18. J. Eder, E. Panagos, and M. Rabinovich. Time constraints in workflow systems. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering*, CAiSE '99, pages 286–300, London, UK, 1999. Springer-Verlag.
19. A. Egyed. A scenario-driven approach to traceability. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 123 – 132, may 2001.
20. A. Egyed, S. Biffl, M. Heindl, and P. Grünbacher. A value-based approach for understanding cost-benefit trade-offs during automated software traceability. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, TEFSE '05, pages 2–7, New York, NY, USA, 2005. ACM.
21. A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *ASE'02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 163, Washington, DC, USA, 2002. IEEE Computer Society.
22. A. Egyed and P. Grunbacher. Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help. *Software, IEEE*, 21(6):50–58, 2004.
23. Alexander Egyed, Florian Graf, and Paul Grunbacher. Effort and quality of recovering requirements-to-code traces: Two exploratory experiments. In *Requirements Engineering, IEEE International Conference on*, pages 221–230, Los Alamitos, CA, USA, 2010. IEEE Computer Society.
24. F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool. In *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard*, pages 97 –105, 2001.
25. V. Gervasi and D. Zowghi. On the role of ambiguity in requirement engineering. In *REFSQ*, pages 248–254, 2010.
26. A. Ghazarian. *A design-rule-based constructive approach to building traceable software*. PhD thesis, Toronto, Canada, 2009.
27. M. D. Gibson and K. Conheeney. Domain knowledge reuse during requirements engineering. In *Proceedings of the 7th International Conference on Advanced Information Systems Engineering*, pages 283–296, London, UK, 1995. Springer-Verlag.
28. B. Gleich, O. Creighton, and L. Kof. Ambiguity Detection: Towards a Tool Explaining Ambiguity Sources. *Requirements Engineering: Foundation for Software Quality*, pages 218–232, 2010.
29. A.F. Gregorc. An adults guide to style. Columbia, CT: Gregorc Associates, Inc. 1, 1982.
30. P. Grunbacher and R.O. Briggs. Surfacing tacit knowledge in requirements negotiation: experiences using easywinwin. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, page 8 pp., 2001.
31. S. Haiduc and A. Marcus. On the use of domain terms in source code. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 113 –122, 2008.
32. T. Hall, S. Beecham, and A. Rainer. Requirements problems in twelve software companies: an empirical analysis. *Software, IEE Proceedings* -, 149(5):153 – 160, October 2002.
33. S. G. Hart and L. E. Stavenland. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In P. A. Hancock and N. Meshkati, editors, *Human Mental Workload*, chapter 7, pages 139–183. Elsevier, 1988.
34. J.H. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 105–115. IEEE, 2005.

35. J. H. Hayes. Building a requirement fault taxonomy: Experiences from a nasa verification and validation research project. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, Washington, DC, USA, 2003. IEEE Computer Society.
36. J. H. Hayes, G. Antoniol, and Y. G. Guéhéneuc. Prereqir: Recovering pre-requirements via cluster analysis. volume 0, pages 165–174, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
37. J. H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *RE '03: Proceedings of the 11th IEEE International Conference on Requirements Engineering*, page 138, Washington, DC, USA, 2003. IEEE Computer Society.
38. J. H. Hayes, A. Dekhtyar, and S. Sundaram. Text mining for software engineering: how analyst feedback impacts final results. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.
39. Jane Huffman Hayes and Alex Dekhtyar. Humans in the traceability loop: can't live with 'em, can't live without 'em. In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, TEFSE '05, pages 20–23, New York, NY, USA, 2005. ACM.
40. J.H. Hayes. Building a requirement fault taxonomy: experiences from a nasa verification and validation research project. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 49 – 59, 2003.
41. W. Jirapanthong and A. Zisman. Xtraque: traceability for product line systems. *Software Systems Modeling*, 8(1):117–144, 2007.
42. J. C. Joseph. Requirements engineering and management: the key to designing quality complex systems. In *The TQM Magazine*, volume 12, pages 400–407. MCB UP Ltd, 2000.
43. E. Kamsties, D.M. Berry, and B. Paech. Detecting ambiguities in requirements documents using inspections. In *Workshop on Inspections in Software Engineering*, pages 68–80, 2001.
44. E. Kamsties, K. H
"ormann, and M. Schlich. Requirements engineering in small and medium enterprises. *Requirements engineering*, 3(2):84–90, 1998.
45. J.E. Kasser. The first requirements elucidator demonstration (FRED) Tool. *Systems engineering*, 7(3):243–256, 2004.
46. B. Kitchenham, O.P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering - a systematic literature review. *Information and Software Technology*, 51:7–15, January 2009.
47. W.W.F. Lau and A.H.K. Yuen. Exploring the effects of gender and learning styles on computer programming performance: implications for programming pedagogy. *British Journal of Educational Technology*, 40(4):696–712, 2009.
48. D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12:359–388, 2007.
49. D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
50. D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3:303–318, 2007.
51. J. Lee and J.Y. Kuo. New approach to requirements trade-off analysis for complex systems. *Knowledge and Data Engineering, IEEE Transactions on*, 10(4):551–562, 2002.
52. A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Adams re-trace: A traceability recovery tool. volume 0, pages 32–41, Los Alamitos, CA, USA, 2005. IEEE Computer Society.
53. A. De Lucia, M. Di Penta, and R. Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering*, 99, 2010.
54. A. De Lucia, M. Di Penta, R. Oliveto, and F. Zurolo. Coconut: Code comprehension nurturant using traceability. pages 274–275, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
55. Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transaction on Software Engineering Methodology*, 16, September 2007.

56. N. Madani, L. Guerrouj, M. Di Penta, Y. G. Guéhéneuc, and Giuliano Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Proceeding of the Conference on Software Maintenance and Reengineering*, pages 69–78. IEEE, 2010.
57. J.I. Maletic, E.V. Munson, A. Marcus, and T.N. Nguyen. Using a hypertext model for traceability link conformance analysis. In *Proceedings of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 47–54, Montreal, Canada, 2003.
58. A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.
59. A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.
60. N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20 –36, 2010.
61. R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 68–71, Washington, DC, USA, 2010. IEEE Computer Society.
62. Y. Park and J.B. Black. Identifying the impact of domain knowledge and cognitive style on web-based information search behavior. *Journal of Educational Computing Research*, 36(1):15–37, 2007.
63. V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, IWPC '02, pages 271–, Washington, DC, USA, 2002. IEEE Computer Society.
64. S. A. Sherba. *Towards automating traceability: an incremental and scalable approach*. PhD thesis, Boulder, CO, USA, 2005.
65. D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.
66. E. Soloway and K. Ehrlich. *Empirical studies of programming knowledge*, pages 235–267. ACM, New York, NY, USA, 1989.
67. G. Spanoudakis, A. Zisman, E. Pérez-Minana, and P. Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105 – 127, 2004.
68. S. K. Sundaram, J. H. Hayes, and A. Dekhtyar. Baselines in requirements tracing. In *Proceedings of the 2005 workshop on Predictor models in software engineering*, pages 1–6, New York, NY, USA, 2005. ACM.
69. M. Taira. The Influence of Domain Knowledge and Task Requirement on the Selection of Learning Strategies in the Internet. *The International Journal of Creativity and Problem Solving*, 18(1):45–53, 2008.
70. A. A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Programming Language*, 4(3):143–167, 1996.
71. W. C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 2003.
72. A. Zisman, G. Spanoudakis, E. Pérez-Miñana, and P. Krause. Towards a traceability approach for product families requirements. In *Proceedings of 3rd ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications, Orlando, USA*, Orlando, USA, 2002.